Technische Universität München
Fakultät für Informatik

Diplomarbeit in Informatik

Hardware Pattern Matching for Network Traffic Analysis in Gigabit Environments

Gregor M. Maier

Aufgabenstellerin:	Prof. Anja Feldmann, Ph. D.
Betreuerin:	Prof. Anja Feldmann, Ph. D.
Abgabedatum:	15. Mai 2007

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum	Gregor M. Maier

Abstract

Pattern Matching is an important task in various applications, including network traffic analysis and intrusion detection. In modern high speed gigabit networks it becomes unfeasible to search for patterns using pure software implementations, due to the amount of data that must be searched. Furthermore applications employing pattern matching often need to search for several patterns at the same time. In this thesis we explore the possibilities of using FPGAs for hardware pattern matching. We analyze the applicability of various pattern matching algorithms for hardware implementation and implement a Rabin-Karp and an approximate pattern matching algorithm in Endace's network measurement cards using VHDL. The implementations are evaluated and compared to pure software matching solutions. To demonstrate the power of hardware pattern matching, an example application for traffic accounting using hardware pattern matching is presented as a proof-of-concept. Since some systems like network intrusion detection systems analyze reassembled TCP streams, possibilities for hardware TCP reassembly combined with hardware pattern matching are discussed as well.

Contents

List of Figures

1 Introduction

The overall goal of this thesis is to explore the possibilities of pattern matching by means of programmable hardware with respect to high speed network environments. A strong emphasis is given to security applications, like network intrusion detection systems (NIDS). This thesis is written in cooperation with Endace [End07] Technology Ltd. in Hamilton, New Zealand. Endace is a global leader in hardware acceleration and network interface technology.

1.1 Motivation

Finding patterns in a large dataset is an important field. Internet search engines, databases, network intrusion detection systems, DNA sequencing, etc. are all examples where pattern matching is employed.

Network intrusion detection systems like Snort [Roe99] and Bro [Pax99] heavily utilize pattern matching in their operation. Another important question is determining the application layer protocol spoken within a connection. Internet service providers are interested in the traffic mix they carry on their links, and intrusion detection systems use application detection to determine which analyzers should be used on the connection. Most systems use a set of well-known ports, such as those assigned by IANA [IANA], or those widely used by convention, to determine the application layer protocol. If a connection does not use one of the recognized ports, this application detection mechanism breaks down. Examples include running a Web server on a non standard port or using port 80 for non Web applications. Pattern matching can be employed to identify application layer protocols without using ports.

Pattern matching in network environments is currently done in software. However, pattern matching algorithms are expensive in terms of CPU utilization, and most applications need to search for several patterns in parallel. In current high speed network environments with several gigabit of network traffic, pure software solutions cannot keep up with the full data rate of network links. Another approach is needed to search for patterns in such environments. On the other hand, pattern matching algorithms are well suited for implementation in programmable hardware, like FPGAs. A short descriptions of FPGAs is given in Appendix B. Furthermore hardware implementations can be parallelized easily. Therefore, hardware implementations are a possible solution for pattern matching problems in gigabit network environments.

Implementing hardware pattern matching in FPGAs can mean two different things. The first approach is to implement a generic pattern matcher in hardware and load this pattern matcher into the FPGA. The patterns to match can then be configured dynamically without the need to load a complete new FPGA image. The FPGA is used as a generic pattern match engine. The second approach is to incorporate the patterns to match into the FPGA logic itself. While the second approach can achieve higher throughput, reconfiguring the pattern set takes a considerable time. Furthermore the second approach is only feasible if a dedicated FPGA is available for pattern matching. If the pattern matching logic shares a FPGA with other logic only the first approach is feasible, because reloading a FPGA image disrupts the operation of the other logic. This thesis uses the first approach, a generic pattern matching engine, where the pattern set can be dynamically reconfigured.

1.2 Related Work

Using FPGAs for hardware pattern matching has been extensively analyzed by several researchers. But these works ([SP01], [CS03], [SP03]) all focus on incorporating the pattern to match into the FPGA fabric itself, rather than implementing a generic match engine in hardware.

Dreger et al. [DFM+06] presented and evaluated a software approach to use pattern matching for application layer protocol detection for the Bro network intrusion detection system. Dharmapurikar and Paxson [DP05] have analyzed hardware TCP stream reassembly for security applications in the presence of adversarial behavior.

Exact pattern matching algorithm is a settled field and most text books on algorithms, like [CSRL01], cover pattern matching in some detail. Approximate pattern matching research is a bit more active. Navarro [Nav01] summarizes current research and the state of the art in approximate pattern matching algorithms.

1.3 About Endace DAG Network Monitoring Cards

This thesis is written with the support of Endace Technology Ltd. Endace is a global leader in hardware acceleration and network interface technology, recognized by an elite, worldwide client base of major corporations, government agencies and ISPs. Endace customers operate the world's largest, fastest and most critical networks. Endace solutions enable global customers to observe and analyze 100% of the traffic carried on their networks, guaranteeing security, integrity and performance for their users and applications, regardless of transmission speed, loading or interface type.

Endace DAG network monitoring interface cards provide 100% packet or cell capture, regardless of interface type, packet size, or network loading. On top of this zero-loss performance, Endace DAG cards offload the data load workload from the

CPU, allowing software applications to inspect and process traffic at much higher speed.

More recent DAG cards also provide hardware-based traffic filtering, data stream replication, and CPU load balancing functions. These effectively accelerate application performance, enabling Network Managers to be confident that every packet has been inspected and reported on, and that nothing has been missed.

1.4 Outline of this Thesis

Chapter 2 describes currently known pattern matching algorithms and then selects two algorithms for implementation in hardware. Chapters 3 and 4 deal with the design and implementation of the selected algorithms in hardware. In Chapter 5 both implementations are evaluated. Chapter 6 describes an example application for hardware pattern matching — the detection of application layer protocols in network traffic using pattern matching. Chapter 7 is dedicated to an initial design of a TCP stream reassembler, that can match complete TCP streams and not just single packets. Finally in Chapter 8 an outlook on future developments and enhancements for hardware pattern matching is given.

2 Pattern Matching Algorithms

Finding all occurrences of a pattern in a text is a problem that arises often. Word processors, search engines, databases, etc. all have to deal with the problem of finding a pattern in a large text. This chapter gives an overview of pattern matching algorithms. First exact pattern matching algorithms are discussed, then an approximate pattern matching algorithm is presented and finally algorithms for hardware implementation are selected.

2.1 Exact Pattern Matching Algorithms

Exact pattern matching algorithms find exact occurrences of a pattern in a text. Let p, x be two strings over a finite alphabet Σ, with $p = p_1 \dots p_m$ and $x = x_1 \dots x_n$. The pattern p occurs in text x if there exists a shift s so that $1 \leq s \leq n - m - 1$ and $x_s \dots x_{s-1+m} = p$. The pattern matching problem is the problem of finding all valid shifts resp. of determining if a valid shift for pattern p and text x exists.

The naive approach to pattern matching checks at every text position $s = 1 \dots n - m - 1$ if $x_s \dots x_{s-1+m} = p$ by comparing character by character and thus requires $O(m \cdot n)$ steps. The more sophisticated algorithms presented in the next sections all do some preprocessing based on the pattern and are able to reduce the running time for pattern matching. A good overview of exact pattern matching algorithms can be found in [CSRL01].

2.1.1 Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm [KMP77] can solve the pattern matching in linear time by preprocessing the pattern. A quick explanation of the algorithm is presented in Wikipedia [Wik07c]. Cormen et al. [CSRL01] present the algorithm too and proof its running time and its correctness.

Consider the following example, were p is the pattern and x is the text to search. Furthermore we keep two variables, s, which donates a possible valid shift and i, an offset into the pattern. Both are initially set to 1.

```
   12345678901234567890123
x: ABC ABCDAB ABCDABCDABDE
p: ABCDABD
   1234567
```

Text and pattern characters are compared and at position 4 a mismatch occurs. The naive algorithm would now continue at text position 1. However, since no `A` occurred in the text between positions 1 and 4, the next possibility for a match is at position 5. Therefore the algorithm sets $s \leftarrow 5$, $i \leftarrow 1$ and continues the search.

```
   12345678901234567890123
x: ABC ABCDAB ABCDABCDABDE
p:     ABCDABD
       1234567
```

The next couple of characters match but at text position 11 the text and pattern characters mismatch again. This time however positions 9 and 10 in the text are a valid prefix of the pattern. The algorithm sets $s \leftarrow 9$ and $i \leftarrow 3$. Since the algorithm already knows that `AB` at position 9 in the text is a prefix of the pattern, the algorithm can continue by comparing x_{11} and p_3.

```
   12345678901234567890123
x: ABC ABCDAB ABCDABCDABDE
p:         ABCDABD
           1234567
```

We observe, that every text character is compared only once, therefore the running time of the actual search is $O(n)$. During the preprocessing phase, the pattern is analyzed and a prefix table is calculated. This prefix table is used during the search phase to determine the appropriate values of s and i. Preprocessing takes $O(m)$ time, therefore the algorithm has an overall running time of $O(m + n)$.

2.1.2 Boyer-Moore Algorithm

The Boyer-Moore algorithm was developed by Bob Boyer and J. Strother Moore in [BM77]. The algorithm can achieve a best case running time of $O(n/m)$ by preprocessing the pattern. The average case running time is $O(n)$, but as opposed to the Knuth-Morris-Pratt algorithm, Boyer-Moore has a worst case running time of $O(m \cdot n)$.

The idea of the Boyer-Moore Algorithm is to skip parts of the text, that cannot match the pattern. At the beginning of the search phase the pattern is left-aligned with the text. The algorithm starts the comparison between text and pattern from right to left by checking if $x_m = p_m$. If the characters match, the algorithm continues from right to left with $x_{m-1} = p_{m-1}$. If a mismatch occurs, two heuristics calculate how far the pattern can be moved to the right, thus skipping over some text positions that cannot yield a match.

The first heuristic is the *Bad-Character-Heuristic*. If a mismatch occurs, the “bad” text character x_b, is searched in the pattern. The pattern is then moved to the right until the last occurrence of the bad character in the pattern is at position b. If the bad character does not occur at all, the pattern is moved its whole length to the right.

The second heuristic is the *Good-Suffix-Heuristic*. If a mismatch occurs while comparing the pattern with the text from right to left and the mismatch is at pattern position i, $i \neq m$, a suffix of the pattern $p_i \dots p_m$ must match, otherwise a mismatch would have occurred earlier. In this case, the pattern is moved to the right until a part word of the pattern matches the suffix again. If no such part word exists, the pattern is moved its whole length to the right.

If the heuristics disagree, the maximum of both is used to move the pattern. A simplified version of this algorithm, the Boyer-Moore-Horspool algorithm, uses only the first heuristic. To achieve a good run time behavior the heuristics are precomputed in an initial preprocessing phase. The Wikipedia [Wik07a] shows an example and pseudo code for the Boyer-Moore algorithm.

2.1.3 Rabin-Karp Algorithm

The Rabin-Karp algorithm was presented by Richard Karp and Michael Rabin in [KR87]. The algorithm achieves an average case running time of $O(m+n)$ by using hashing. The worst case running time is still $O(m \cdot n)$ however. The Rabin-Karp algorithm exploits the fact that if two strings are equal, their hash values are also equal. But since two hash values can be equal even if the underlying strings differ, the algorithm has to verify every match of hash values. In a preprocessing phase the hash value of the pattern p is calculated. Then for every text position $s = 1 \dots n - m - 1$ the hash value of $x_s \dots x_{s-1+m}$ is compared to the hash value of the pattern. If the hash values match, the pattern is compared to the text at position s to verify that there is indeed a match at this position. The crucial point is the calculation of the hash values of the text. If the complete hash value has to be recalculated for every text position the running time of the algorithm would be the same as the running time of the naive approach. A rolling hash function is therefore employed. A rolling hash function can compute the hash value of $x_s \dots x_{s-1+m}$ in constant time from the characters x_{s-1}, x_{s-1+m}, and the previous hash value, $x_{s-1} \dots x_{s-2+m}$.

Another advantage of the Rabin-Karp algorithms is the fact that it can be used to efficiently search for multiple patterns in a text. In the preprocessing phase all patterns are hashed. Then the algorithm checks for every text position if the current text hash value matches any of the pattern hashes. It must be noted however, that the hash value depends on the length of pattern, therefore a separate text hash must be calculated for every pattern length that should be matched.

2.2 Approximate Pattern Matching Algorithms

A problem in network monitoring is dealing with deviations or errors in the data stream. Bit errors might have been introduced into the data or an attacker might have inserted a backspace to obscure the pattern of the attack. Other areas were deviations and errors of data occur are finding a pattern in the presence of typing or spelling errors or searching for DNA sequences. The goal of *approximate pattern matching* is to match patterns in a text where the pattern and/or the text contain a limited number of such deviations. A good overview of approximate pattern matching algorithms and current developments in this area can be found in [Nav01].

In order to decide if a pattern is contained in a text with a limited number of errors, a metric for measuring errors must be employed. The most common metric is the *edit distance* [Lev66]. The edit distance between two strings is the minimum number of insertions, deletion, and substitutions of single characters required to translate the first string into the second string. A separate cost could be assigned to each operation (insert, delete, substitute) but for most purposes a uniform cost of 1 per operation is sufficient.

Sellers [Sel80] introduced the first algorithm for this problem which uses dynamic programming. The general principle of this algorithm is still used in current implementations although various improvements have been suggested to increase the algorithm's performance. In the following the algorithm to calculate the edit distance between two strings is presented. This algorithm will then be used to construct an approximate pattern matching algorithm.

2.2.1 Calculating the Edit Distance

Let p, x be two strings, with $p = p_1 \dots p_m$ and $x = x_1 \dots x_n$. The algorithm computes a $(m+1) \times (n+1)$ matrix C, where $C_{i,j}$ corresponds to the minimum number of edit steps to transform $p_{1\dots i}$ to $x_{1\dots j}$. The matrix cells are recursively defined as

$$\begin{aligned}
C_{i,0} &= i \\
C_{0,j} &= j \\
C_{i,j} &= \begin{cases} C_{i-1,j-1} & \text{if} \quad p_i = x_j \\ 1 + \min\left(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}\right) & \text{else} \end{cases}
\end{aligned}$$

for all $i \in 0 \dots m$ and $j \in 0 \dots n$. After the matrix has been evaluated the last cell $C_{m,n} = ed(p,x)$, with $ed(p,x)$ being the edit distance between p and x.

Induction basis: The value of $C_{i,0}$ is the edit distance between the string $p_{1\dots i}$ and the empty string, $C_{j,0}$ is the edit distance between $x_{1\dots j}$ and the empty string. To transform $p_{1\dots i}$ to the empty string, i deletions are required.

		h	a	l	l	w	a	y	s
	0	1	2	3	4	5	6	7	8
h	1	0	1	2	3	4	5	6	7
a	2	1	0	1	2	3	4	5	6
l	3	2	1	0	1	2	3	4	5
f	4	3	2	1	1	2	3	4	5
w	5	4	3	2	2	1	2	3	4
a	6	5	4	3	3	2	1	2	3
y	7	6	5	4	4	3	2	1	2

Figure 2.1: Edit distance matrix between “hallways” and “halfway”

Induction step: Assume that for two none empty strings of lengths i and j all edit distances for their prefix strings have already been computed. The goal is to transform $p_{1...i}$ to $x_{1...j}$. If the characters p_i and x_j are equal, the edit distance is the same as for $p_{1...i-1}$ to $x_{1...j-1}$. If p_i and x_j differ, there are three possibilities:

- p_i can be deleted. The resulting edit distance is thus 1 plus the edit distance between $p_{1...i-1}$ and $x_{1...j}$.
- x_j can be inserted at the end of $p_{1...i}$, resulting in an edit distance of 1 plus the edit distance between $p_{1...i}$ and $x_{1...j-1}$.
- p_i can be substituted with x_j, resulting in an edit distance of 1 plus $p_{1...i-1}$ and $x_{1...j-1}$.

The $\min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$ term in the above formula selects the operation that leads to the smallest edit distance.

The matrix can be computed row-wise or column-wise. Figure 2.1 shows the edit distance matrix between ”hallways” and ”halfway”. The runtime of the algorithm is $O(mn)$, but the space requirement is only $O(\min(m, n))$. At any point, only the last column (resp. the last row) must be stored to calculate the new column (row). It is also possible to extract the sequence of operations used to transform p to x by traversing the matrix from $C_{m,n}$ to $C_{0,0}$ while writing down the operations. Multiple such paths may exist.

Ukkonen [Ukk85] pointed out some interesting properties of the above matrix, that allow algorithms that are more efficient. In particular he showed that neighbor cells differ by at most 1. He also analyzed the data dependencies between cells. Indeed he showed that it is possible to evaluate the matrix diagonal-wise (either primary or secondary diagonals).

		h	a	l	l	w	a	y	s
	0	0	0	0	0	0	0	0	0
h	1	0	1	1	1	1	1	1	1
a	2	1	0	1	2	2	1	2	2
l	3	2	1	0	1	2	2	2	3
f	4	3	2	1	1	2	3	3	3
w	5	4	3	2	2	1	2	3	4
a	6	5	4	3	3	2	1	2	3
y	7	6	5	4	4	3	*2*	*1*	*2*

Figure 2.2: Finding "halfway" in "hallways" with two errors

2.2.2 Finding Approximate Matches

The approximate pattern matching algorithm is basically the same. Only the fact that a match can start at any text position must be accounted for. This is achieved by setting $C_{0,j} = 0$, for all $j \in 0 \ldots n$. I.e., the empty pattern matches with zero errors an any text position. The text is processed character by character and for each character x_j the according column of the matrix is calculated. A match is found at text position j if $C_{m,j} < k$, where k is the maximum number of errors allowed. Figure 2.2 shows the matrix when searching for the pattern "halfway" in the text "hallways". Italic entries are the positions were a match with less than 2 errors was found. Over the years various improvements to this algorithm have been developed. Most of these exploit properties of the dynamic programming matrix.

Another approach for solving the approximate pattern matching problem is using automatons, where each specifity of a column represents a separate state of the automaton. In this case, state transitions occur on every character of the text, i.e., when a new column is computed. All these improvements achieve increased performance by trading memory for runtime. Navarro [Nav01] summarizes these improvements.

2.2.3 Limited Expressions

An interesting extension to the algorithm allows to match *limited expressions* with errors. A limited expression is similar to a regular expression, but without the variable length modifiers $(*, +, \{\})$. Therefore, character classes (e.g., $[0-9]$, $[a-fA-F]$) and simple wildcards (.) can be specified in limited expressions. This also implies that a limited expression pattern always has a clearly defined length.

In order to use limited expressions, the comparison $p_i = x_j$ is replaced with a table lookup. A separate table for every byte of the pattern is required. The tables contain one entry per character, i.e., 256 entries for ASCII. A table entry is a boolean value which determines if the associated character matched. To compare p_i to x_j, the character x_j is looked up in the table for pattern byte i by using x_j as index into the table. If the lookup yields *true*, the character x_j matches the position i of the pattern (either the characters are the same or a wildcard or character class at position i matched). No other changes to the algorithm are required.

2.3 Selecting Algorithms for Hardware Implementation

The goal of this thesis is to identify an algorithm and an implementation that can keep up with the full data rate of the network link, regardless of the burstiness of the traffic, the size distribution of packets, or any other factors.

The length, n, of the text to search is normally the largest factor that determines the runtime of a pattern matching algorithm. A software solution would pick an algorithm that minimizes the impact of the text length n on the runtime. However for the application targeted in this thesis, doing hardware based pattern matching on online network traffic, dependencies on the text length (i.e., the volume of network traffic) are not important. The network traffic flows through the match engine in real time, so the important factor is the amount of work *per byte of network traffic*. After all the hardware must be able to keep up with the speed of the network.

Another key criteria is how easily one can parallelize the algorithm. Most applications are searching for several patterns at the same time, therefore a hardware pattern matcher must be able do to so.

The final important criteria is the amount of memory required for a matcher. On-chip memory on FPGAs is very limited, only a couple of KB are available, therefore an algorithm should not require much memory. Furthermore dynamically allocating memory is a hard task in hardware, so ideally an algorithm for hardware implementation should only use a predefined, fixed amount of memory.

The Rabin-Karp algorithm has been selected, because it is able to handle large numbers of patters and lends itself to be parallelized. The rolling hash can be implemented easily in hardware and the comparison of the current text hash with the pattern hashes can be achieved by CAM like lookup tables. The number of patterns that can be matched in parallel is only limited by the size of the lookup table. The Rabin-Karp algorithm can be used to match hundreds of patterns in parallel.

The hardware matcher will report a possible match whenever the pattern hash and the current text hash match. A software application must then verify if the reported match is indeed a real match or if it is a false positive. As long as the number of false positives is low, the software is able to verify all reported matches and keep up with the data rate. Implementing the verification in hardware is in principle possible.

But it would not be possible to give performance guarantees, since in worst case every text byte can result in a match that must be verified.

The advantage of the Rabin-Karp algorithm is that it can match a vast number of patterns in parallel. It has the disadvantage that some applications, like NIDSes, have to match patterns with wildcards and they may also want to account for errors in patterns. E.g., to prevent evasion attacks, when attackers insert characters to confuse pattern matching algorithms. Rabin-Karp can only match exact patterns without wildcards or errors. Approximate pattern matching with limited expressions enables such sophisticated matches, but the approximate pattern matching algorithm cannot be parallelized as well as Rabin-Karp, therefore the number of patterns that can be matched in parallel is much smaller.

The other classical pattern matching algorithms, Boyer-Moore and Knuth-Morris-Pratt cannot be parallelized as easily as Rabin-Karp. A separate matching unit would be required for every pattern, thus limiting the number of patterns that can be handled. Furthermore implementing an approximate pattern matcher does not require significantly more resources than implementing Boyer-Moore or Knuth-Morris-Pratt.

Since there is no algorithm that can achieve everything, both, the Rabin-Karp and an approximate pattern matching algorithm have been implemented. This gives applications the choice of an appropriate matcher for their specific needs.

3 Design and Implementation of the Rabin-Karp Algorithm

In this chapter the design and implementation of the Rabin-Karp algorithm is described. After an introduction, the hash function to use will be selected, the implication of using wide data paths is discussed, the hash lookup mechanism is described and finally the actual hardware implementation is presented.

The hardware based Rabin-Karp pattern matcher enables a software application to match a larger number of patterns at line rate. For example the Snort [Roe99] intrusion detection system uses several hundred patterns to search network traffic for attack patterns. The Rabin-Karp matcher is able to speed up such applications by offloading the pattern matching onto hardware. The application software, e.g., Snort, loads the patterns into the DAG cards. It is possible to modify the set of patterns in the card while capturing is in progress, although packets are not matched while the pattern set is being changed.

The pattern matcher can mark packets containing matches by overwriting parts of the ERF [End04] header. It can also discard packets. The application software can configure whether packets should be marked and/or discarded and if discarding is enabled, the software can configure if matched or unmatched packets shall be discarded. The mark contains the information which *pattern lengths* had a match, not the actual pattern that had a match. When the software receives a network packet that is marked, it can run a software based pattern matcher on the packet to decide if the mark is a false positive and also to decide which pattern matched. Snort for example does not match all patterns in all packets. The patterns that are matched against a packet depend on the IP addresses and port numbers and on the position of the packet in the connection. E.g., some patterns must occur at the beginning of a connection in order to match. Using this premises, the verification time can be reduced, since the application software only has to verify a match if it occurred in an appropriate packet.

3.1 Design Overview

The general principle is that for every byte of network data (respectively in every clock cycle), the new hash value is calculated and this hash value is then looked up in a table containing the hash values of all patterns. If the hash is found in the table, the pattern is a possible match. If marking packets is enabled, the appropriate mark information is written into the ERF header. False positives can occur due to hash collisions.

Since the hash function and the hash value depend on the length of the pattern, a separate hash calculator and a separate lookup table per pattern length is required.

3.2 Selecting a Hash Function

The hash function must be easy and fast to calculate. Furthermore an ideal hash function should be a rolling hash, i.e., a new hash value can be computed by using the old hash value, the current byte of packet data, and the byte of packet data seen "pattern length" byte before. The xor operation ($\oplus$) is a suitable function for this purpose. The hash value can then be calculated as

$$hash \leftarrow hash \oplus packet[curidx] \oplus packet[curidx - patternlen]$$

where *packet* is the packet data, *curidx* is the current position within the packet, and *patternlen* is the length of the current pattern.

Using the packet byte directly however, will lead to a lot of collisions and false positives. To increase the entropy the current byte is used as an index into a table containing 256 randomly generated and uniformly distributed 32 bit values. The xor is then calculated using these derived 32 bit values. The lookup table can be implemented using dual port BlockRAM, in a 32bit×512 entries configuration. The lookup In the following the lookup operation is denoted as $ht[x]$.

Just xoring the 32 bit values still leads to a large number of false positives. The problem is that $x \oplus x$ is always 0. This is a problem for odd and even pattern lengths. For an even pattern length, the total hash of a pattern may become 0. For an odd pattern length it may become $0 \oplus ht[x]$, i.e., the hash may be reduced to the value of just one byte.

One solution is to rotate some of the 32 bit input "bytes" before xoring them. The disadvantage that arises now is, that the hash function is no longer a completely rolling hash, since some of the 32 bit values are rotated. Tests showed that rotating only the first seven input "bytes" gives us a good trade-off between performance of the hash function and number of false positives. The formula to calculate the new hash value thus becomes:

$$\begin{aligned} hash \leftarrow\ & \mathsf{ror}(ht[packet[curidx]], sh1) \\ & \oplus \mathsf{ror}(ht[packet[curidx-1]], sh2) \\ & \oplus \ldots \oplus \mathsf{ror}(ht[packet[curidx-7]], sh7) \\ & \oplus ht[packet[curidx-8]] \\ & \oplus \ldots \oplus ht[packet[curidx - patternlen] \end{aligned}$$

where ror is bit rotate right and $sh1 \ldots sh7$ are the numbers of bits the values are rotated. The numbers of bits to rotate are chosen randomly. The values should not be multiples of each other. For example, shift values of 4, 8, 12, 16, etc. are a bad choice. To calculate the value of $ht[packet[curidx - 8]] \oplus \ldots \oplus ht[packet[curidx - patternlen]]$ the rolling hash approach can be used.

3.3 Using Wide Datapaths

DAG network monitoring cards do not process network packets byte-wise. They use a wide datapath of several bytes to increase throughput. Therefore the hardware pattern matcher must be able to process several bytes in every clock cycle. This implies, that there are several bytes in every cycle where a match for a pattern can start and the pattern matcher must therefore be able to calculate several hashes per pattern length and cycle, each one starting at a different byte of the current input word.

3.4 Hash Lookups

The lookup table can be implemented using a CAM. On a 32 bit datapath 4 lookups per cycle and pattern length are required, therefore 4 CAMs are be required per pattern length. This CAM must be implemented completely inside the FPGA, since no external CAM resources are available. The Xilinx CoreGen tool can generate 32 bit wide CAMs with 32 entries, but such a CAM requires 4 BlockRAMs. Furthermore these CAMs only provide one search port. Therefore 16 BlockRAMs per pattern length are required. This is way too much, since the FPGAs used only feature a total of 100 to 200 BlockRAMs and a significant number of these are already used for other components of the DAG card.

To overcome this constraint a *fuzzy CAM* has been developed. The fuzzy CAM exploits the fact that false positives are acceptable and that all CAMs needed for one pattern length contain the same data. Furthermore a real CAM returns the location of the entry, a fuzzy CAM on the other hand will only indicate if the search term was found in the CAM or not, but not where it was found. This means that the pattern matcher will not be able to determine which pattern matched, it will only be able to determine that *a* pattern matched. Since the application software has to verify matches anyway and since this restriction yields a CAM that uses much less BlockRAMs, the restriction is an acceptable trade-off.

A design in which the hardware matcher can inform the software application exactly which pattern had a potential match would be possible however. If a fuzzy CAM detects a match, the 32 bit hash value that resulted in the match could be included in the ERF header, so that the application software can analyze it. However this requires additional bandwidth, since the pattern matcher must insert the hash values into the data stream. Furthermore many hash values are looked up every cycle (datapath width in bytes $\times$ number of pattern lengths) and each of those can result in a match. If more than one such match occurs in one clock cycle, an arbiter and queues are be required to be able to include all matches in the ERF header. Implementing an arbiter in hardware is complex and requires a significant amount of logic resources. Since the downsides of such a design outweigh potential gain, the hash values are not included in the ERF header.

As mentioned in Section 2.3, the match verification is not done in hardware. If match verification would be employed, the implementation could not guarantee line rate matching anymore. Furthermore match verification can be done easily in software, as long as the hashing does only yield a small number of false positives.

3.5 Hardware Implementation

Two DAG network monitoring cards were chosen as target system for the Rabin-Karp implementation,

- DAG 4.5, gigabit ethernet card, with a Xilinx VirtexII Pro 30 FPGA. This FPGA contains approximately 13700 slices and 136 BlockRAMs, so a total of 27400 LUTs and 27400 Flip Flops are available. A description of the internal architecture of FPGAs is available at [Wik07b] and a short introduction is presented in Appendix B.
- DAG 8.2X, 10 gigabit ethernet card, with a Virtex4 SX35. This FPGA contains approximately 15400 slices and 192 BlockRAMs.

The number of different pattern length that the pattern matcher can handle is only limited by the available resources in the FPGA, especially the number of available BlockRAMs is the major bottleneck. For both cards a number of 6 different pattern lengths was found to be feasible. Which length are used is arbitrary. For the current implementation pattern length of 4, 5, 6, 7, 8, and 9 bytes were chosen, since most patterns used by systems like Snort are rather short and longer patterns can be truncated. There is no technical limit to the maximum pattern length.

The implementation of the Rabin-Karp pattern matcher has been tested and used on datapaths up to 64 bit wide with a maximum maximum clock frequency of 200 MHz. Since the pattern matcher can process one input word every clock cycle, a maximum bandwidth of 12.8 Gbit/s can be achieved.

4 Design and Implementation of an Approximate Pattern Matcher

This chapter covers the design and implementation of an approximate pattern matcher. After an introduction, the comparison of pattern and packet data is described, then the approach to calculate the edit distance matrix and the encoding used to represent the values in the matrix are discussed. Finally the actual hardware implementation is presented.

The hardware based approximate pattern matcher enables a software application to match up to 24 different limited expression (see Section 2.2.3) at line rate. Furthermore it can match patterns that contain a defined number of deviations. For example the Bro [Pax99] IDS uses regular expression patterns for its analysis. Although limited expressions are only a subset of regular expressions, the approximate pattern matcher can be configured with patterns that match a superset of the regular expressions used by Bro. If a match is report by the hardware matcher, the Bro system can then run a software based regular expression matcher to verify if the match reported by hardware is indeed a match for the regular expression. Thus the approximate hardware pattern matcher can greatly enhance the system's performance by reducing the number of packets Bro has to analyze.

The application software loads the patterns to match into the card and specifies the maximum number of deviations allowed for each pattern. The patterns loaded can be modified during a capture session, although the pattern that is modified will be disabled while the reprogramming is in progress. I.e., this pattern will not match any packet during this time. However the other patterns are not affected by the reconfigure operation.

Similar to the Rabin-Karp matcher, packets containing matches can be marked by overwriting parts of the ERF [End04] header. The mark contains the information *which* pattern matched.

4.1 Comparing the Pattern and Packet Bytes

In order to utilize limited expression matching, packet bytes are presented to a lookup table instead of comparing them directly to the pattern. As pointed out in Section 2.2.3, one lookup table is required for every byte of the pattern. Every lookup table yields a boolean result and all lookups for the current packet byte must be processed in one clock cycle. I.e., x_i must be compared to all pattern bytes j,

$j \in 1 \ldots m$ during one clock cycle. Where x_i is the current byte of packet data and m is the length of the pattern. These m lookups can be combined, since the index used for all these lookups is the byte x_i. The lookup yields a m bit value, where bit j corresponds to the comparison of x_i to p_j. These tables are implemented using Xilinx BlockRAMs.

4.2 Computing the Edit Distance Matrix

Computing the edit distance matrix C column-wise is not feasible, because the last row of each column depends on every other cell in this column. For implementation in DAG cards, with their several byte datapaths one new column per datapath byte would have to be computed every cycle, and the cell in the last row and column depends on all other cells. For a pattern length of 16 bytes, more than a 100 cells can influence the value of this last cell. It is impossible to compute all these values in one clock cycle.

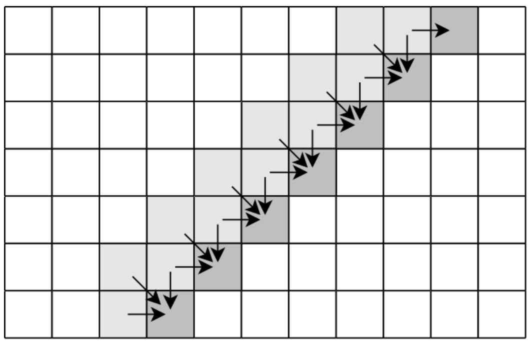

Figure 4.1: Data dependencies when evaluating the edit distance matrix secondary-diagonal wise

Ukkonen [Ukk85] analyzed the dependencies involved in the computation of the edit distance matrix. This leads to an evaluation strategy, where the matrix is computed secondary-diagonal wise. With this computation strategy, the values in the new diagonal only depend on values, that have been calculated in the two previous cycles. Figure 4.1 illustrates this. Dark-gray cells denote the diagonal that is going to be calculated and the arrows indicate the cells (colored light-gray) that influence these new cells. Of course, on wide datapaths more dependencies exist. Several input bytes have to be handled in one clock cycle. But the number of data dependencies is now small enough that the new diagonals can be computed in one clock cycle even for a 32 bit datapath. Still there are too many dependencies for wider datapaths.

Computing the distance matrix secondary-diagonal wise has a small disadvantage. The value in the last cell $C_{m,n}$ is only available m cycles after the last byte of data has been processed. This must be taken into account when implementing the control logic for the pattern matcher.

4.3 Encoding the Cells of the Edit Distance Matrix

Another issue is the term $1+\min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$. Calculating the minimum involves comparing and therefore subtracting the values from each other. Subtractions are an expensive operation due to the carry logic involved. Storing the cells of the edit distance matrix as ordinary binary numbers and using binary arithmetic to calculate the minimum and for doing the +1 addition is therefore suboptimal.

In the approximate pattern matcher implementation a faster method is used. A bit field is used to encode the numbers in the matrix. To represent a zero, all bits are set to '0', the number of bits set to '1' at the right end of the bit field indicate the current value of the edit distance. E.g., 00011 represents two, 00111 three. This implies that there is one and only one transition from 0 to 1 when examining the bit field from left to right.

Calculating the minimum of a any number of values is now easy. The bitwise and of all values corresponds to the minimum — it has only those bits set to '1' that are '1' in every comparand. Adding 1 to the value is easy too. The value is shifted to the left and a '1' is shifted in. Bitwise and and shifts are fast operations in hardware.

Determining if an edit distance is smaller than the maximum allowed is easy too. The maximum is encoded the same way as the edit distance and then the one's complement of it is stored. The edit distance and the maximum are anded. If the result is all '0' the edit distance is lower or equal than the maximum and a match has been found. Consider the following example. The maximum allowed edit distance is 2, which translates to 00011, respectively 11100 after the one's complement. If the calculated edit distance is 3 (00111) the and product is 00100, which indicates that there were too many errors. If the calculated edit distance is 2 (00011) however, the and product is 00000, which indicates that the pattern matched with at most two errors.

4.4 Hardware Implementation

Unfortunately the approximate pattern matcher cannot be parallelized easily. The only way to match several patterns in parallel is to have multiple match units each of which uses a considerable amount of FPGA resources.

Two DAG network monitoring cards were used to implement the approximate pattern matcher:

- DAG CoProcessor expansion card, usable with DAG 4.3 cards. The CoProcessor card has a Xilinx VirtexII 2000 FPGA. This FPGA contains about 10750 slices and 56 BlockRAMs, so about 21500 LUTs and 21500 Flip Flops are available.
- DAG 4.5, gigabit ethernet card, with a Xilinx VirtexII Pro 30 FPGA. See Section 3.5 for the available logic resources on this FPGA.

A trade-off has to be made between resource requirements per matcher and maximum pattern length and maximum number of allowed errors. The amount of logic resources required scales with the product of both parameters. The maximum pattern length was defined as 16 bytes and the maximum number of errors as 4. The actual number of errors per pattern is still user configurable. Most patterns used by Bro for dynamic protocol detection and botnet detection [DFM+06] are shorter than 16 bytes. Patterns longer than 16 bytes can be truncated to 16 bytes. The maximum number of errors per pattern is defined as 4, because that is a fourth of the maximum pattern length.

The number of parallel pattern matchers is limited by the available hardware resources. On the CoProcessor card almost the complete FPGA can be used to implement pattern matchers, allowing the placement of 24 parallel match units.

The approximate pattern matcher implementation can handle a 32 bit wide datapath with a maximum clock frequency of 200 MHz. Since the pattern matcher can match at full data rate, a bandwidth of 6.4 Gbit/s can be achieved.

5 Evaluating the Hardware Pattern Matcher

In this chapter the Rabin-Karp and approximate pattern matching hardware implementations are evaluated. Since the Rabin-Karp matcher only compares the hash values but does not verify the results, it is important to know how this match verification will affect an application software utilizing the Rabin-Karp matcher. Furthermore it is interesting to know how a pure software implementation compares to out hardware implementation. Another question is which throughput is achievable by pure software solutions? To quantify these figures, software only implementations for the Rabin-Karp and the approximate pattern matcher are used and their runtime is analyzed.

5.1 Rabin-Karp

To evaluate the performance of the Rabin-Karp pattern matcher, test patterns are extracted from the Snort [Roe99] intrusion detection system. This extraction yields about 1800 patterns. The Rabin-Karp has support for 6 different pattern lengths, 4, 5, 6, 7, 8, and 9 bytes. Snort patterns that are longer than the maximum pattern length of 9 bytes are truncated to one of the supported lengths.

All tests are done in software, since evaluation of the match and false positive rates require software verification anyway. Furthermore simulating hardware behavior in software allows us to verify the actual hardware implementation. The speed comparison has to be done in software anyway. The tests are done with 50 patterns per length and with 250 patterns per length, i.e., the total number of patterns for the tests are 300, resp. 1500 patterns. These numbers where chosen to determine the influence of the number of patterns on the false positive rate. Furthermore 1500 is used as upper limit to reduce the runtime of the tests. These 300, resp. 1500 patterns are selected randomly from the extracted patterns. Furthermore several sets of patterns are used for the tests with 300 patterns, each set selected from the Snort patterns.

The data on which the pattern matching is applied is a full network trace, including all packet payloads. The trace was collected in 2001 at Auckland University. It contains approximately 23 GB of data in 50.8 million packets. About 19 GB of the trace file is application layer payload data.

All tests were repeated multiple times to rule out deviations and measurement errors.

set	num	base	match	real match	FP to total	FP to real
	1500	Pkts	22.5%	21.8%	0.71%	3.24%
A	300	Pkts	3.9%	3.9%	0.01%	0.31%
B	300	Pkts	5.6%	5.5%	0.12%	2.18%
C	300	Pkts	11.0%	11.0%	0.01%	0.13%
D	300	Pkts	6.5%	6.4%	0.01%	0.20%
	1500	Bytes	40.0%	38.2%	1.81%	4.75%
A	300	Bytes	8.8%	8.8%	0.03%	0.35%
B	300	Bytes	13.2%	13.0%	0.15%	1.18%
C	300	Bytes	20.0%	19.9%	0.04%	0.18%
D	300	Bytes	13.4%	13.4%	0.03%	0.20%

Figure 5.1: Match and false positive (FP) rates for Rabin-Karp

5.1.1 Match and False Positive Rate

The Rabin-Karp implementation will generate false positive matches. Therefore software applications utilizing the hardware matching must verify every reported match. It is import to know how many packets match, i.e., how many packets must be verified in software. Furthermore the number of false positives is also important to determine the overhead incurred due to false positives.

Figure 5.1 shows the match and false positive rates. Match and false positive rate are noted in percent of total number of packets and in percent of the number of bytes. The number of bytes used is based on IP volume, as specified by the *IP total length* header field of IP packets. Link layer headers have been omitted from these numbers.

The *set* column denotes the pattern set used, *num* the number of patterns, and *base* specifies whether the figures are based on packets or bytes. *match* is the rate of packets, resp. bytes matched by the hardware matcher in percent and *real match* is the match rate after software verification, i.e., the true number of matches. The false positive rates are calculated against the total number of packets resp. bytes processed (column *FP to total*) and against the number of packets resp. bytes of real matches (column *FP to real*).

The rate of false positives is low. Less than 2% of the total traffic (5% of matched traffic) were falsely matched. When searching for only 300 patterns these rates drop to 1% or less.

It can be seen that even when searching for 1500 patterns, about 60% of the traffic volume does not result in a match, i.e., a software application like Snort has to verify only the remaining 40% of traffic. These numbers drop significantly to just 13% to 20% when looking for only 300 patterns. It can also be seen, that these match rates depend heavily on the selected patterns. Pattern set C has a match rate twice as high as the other sets and set B has a much higher false positive rate. As expected, the match rates on a byte basis are much higher, since larger packets are more likely to result in a match.

5.1.2 Speed Comparison

In order to quantify the performance gained by using a hardware matcher, the time required to do the matching in software is analyzed. The software matcher is run with and without verification to quantify the time needed for match verification. It must be noted however, that the software matcher uses a different hash function than the hardware matcher. Therefore the number of false positives may differ and the time needed for verification may differ between hardware and software implementations.

No free Rabin-Karp library could be found, therefore the tests were done using a self written Rabin-Karp pattern matcher. While the implementation is correct, it can be assumed that it is possible to further optimize the Rabin-Karp software implementation.

The computer used to calculate these results is an Intel Pentium 4 Xeon with 2.8 GHz and 1 GB of RAM. There are no significant deviations when using different pattern sets, therefore average values are shown.

Figure 5.2 depicts the results of the measurements. The table shows the CPU time spend in userspace, as reported by the time program. Furthermore the resulting data throughput is calculated and displayed. The *rate all* column denotes size of the total trace divided by the time needed, and the *rate PL* column denotes the size of application layer payload divided by time needed. The *verify* column indicates whether match verification was turned on or off, the rows labeled *diff* show the time difference between the runs with and without verification. The *num* column indicates the number of patterns.

Running a Rabin-Karp pattern matcher completely in software is slow. It is unlikely that better optimization would be able to increase the throughput into the gigabit range. The time needed for match verification however is reasonably low. If an application software only verifies matches reported by a hardware matcher, throughput of 1 to 2 Gbps are achievable while searching for 300, resp. 1500 patterns in parallel.

These figures illustrate that the approach presented in this thesis, a Rabin-Karp hardware pattern matcher with match verification in software, is a feasible solution in gigabit network environments.

verify	num	time	rate all	rate PL
no	300	3624s	45.9 Mbps	41.5 Mbps
yes	300	3688s	45.1 Mbps	40.7 Mbps
diff	300	64s	2596 Mbps	2348 Mbps
no	1500	3756s	44.2 Mbps	40.0 Mbps
yes	1500	3902s	42.6 Mbps	38.5 Mbps
diff	1500	246s	1138 Mbps	1029 Mbps

Figure 5.2: CPU time required for Rabin-Karp pattern matching in software

5.2 Approximate Pattern Matching

Analyzing the match rate of the approximate pattern matcher is not required, since unlike the Rabin-Karp matcher, the approximate matcher does not yield any false positives.

The interesting figure is the amount of CPU time it takes a pure software implementation to do the approximate matching. The TRE library [TRE] and the PCRE library [PCR] where considered for this task. The TRE library does approximate regular expression matching. The PCRE library does Perl compatible regular expression matching but does not allow for approximate matching. While there are many other libraries for regular expression matching, no other library for approximate limited respectively regular expression could be found.

In their default configuration both TRE and PCRE are approximately equally fast. However the PCRE library supports a *pcre_study()* function call to optimize an expression. Using this function increases PCRE's speed almost eightfold. When using approximate matching with TRE its performance decreases several orders of magnitude. Since PCRE is that much faster than TRE, PCRE is used for comparison. To be able to get measurements with approximate matching turned on, a very simple, self written approximate pattern matcher with limited expressions is used for this purpose. This matcher is faster than TRE, although it only uses the basic algorithm without any of the performance improvements.

Another question is, how multiple patterns can be matched in software. One approach is to match every packet against every pattern and the second approach is to combine all patterns into one large pattern by using the branch operator |.

The patterns used for the evaluation are the ones that are used in Section 6 for application protocol detection. Since PCRE cannot do approximate matching, only exact matches were searched for when using PCRE. To also see the scalability of these software implementations, the measurements where done with 9 and with 24 patterns. All measurements have been repeated several times. There was no significant deviation.

Pattern Matcher	**branch**	**num**	**time**	**rate all**	**rate PL**
PCRE	yes	9	533s	311 Mbps	281 Mbps
PCRE	no	9	722s	230 Mbps	208 Mbps
apm	n/a	9	8046s	21 Mbps	18.7 Mbps
PCRE	yes	24	2052s	81 Mbps	73 Mbps
PCRE	no	24	1888s	88 Mbps	80 Mbps
apm	n/a	24	21765s	8 Mbps	7 Mbps

Figure 5.3: CPU time required for approximate limited expression matching in software

The computer used to calculate these results is an Intel Pentium 4 Xeon with 2.8 GHz and 1 GB of RAM. The network trace used is the same as for the Rabin-Karp algorithm, i.e., a 23 GB full trace with approximately 50.8 million packets. The matcher has only been applied to the packet payload, thus reducing the size to match to approximately 19 GB.

Figure 5.3 depicts the results of the measurements. The table shows the CPU time spend in userspace, as report by the time program. Furthermore the resulting data throughput is calculated and displayed. The *rate all* column denotes size of the total trace divided by the time needed, and the *rate PL* column denotes the size of application layer payload divided by time needed. The *num* column indicates the number of patterns, and the *branch* column denotes whether the branch operator or separate patterns are used.

The maximum throughput that could be achieved was only 311 Mbps and that was by using only 9 different patterns. For regular expression matching, the length of a pattern influences the runtime quadratically. This is the reason why using the branch operator for PCRE is faster than separate patterns with 9 patterns, but slower with 24 patterns. The approximate software matcher is quite slow compared to PCRE. Using the speed improvements available for approximate pattern matchers might yield a speed-up for the approximate matcher.

6 Application Detection Using Hardware Pattern Matching

In this section an example application using approximate hardware pattern matching to analyze and detect the traffic mix on network links is presented as a proof-of-concept. We limit the application detector to a proof-of-concept implementation, because a final implementation would require a significant amount of profiling work to review the used pattern set and the application layer protocol catalogue we try to detect. Such profiling is a full project of its own, which goes beyond the scope of this thesis.

Applications like network intrusion detection systems or traffic accounting applications need to now which application layer protocol is spoken within a connection. Most systems use a set of well-known ports, such as those assigned by IANA [IANA], or those widely used by convention. If a connection does not use one of the recognized ports, this application detection approach breaks down. Examples include running a Web server on a non standard port or using port 80 for non Web application. Some recently developed application layer protocols are in fact designed not to use fixed ports for their operations. Important examples include the voice-over-ip software Skype [BS06] or file sharing protocols.

In [DFM+06] Dreger et al. presented and evaluated an approach using pattern matching for protocol detection. Their application detector is implemented in the Bro NIDS. The hardware pattern matching implementations presented in this thesis are not yet appropriate for use in intrusion detection. The Bro NIDS for example matches patterns on *reassembled* data streams. If matching is done on a per packet basis only, attackers would be able evade detection by splitting the compromising patterns across packet boundaries. For security applications, like NIDS, this must be accounted for. The packet based hardware pattern matching implementation can be used to speed up matching however. Since the packets themselves are matched by the hardware matcher, the NIDS only has to do additional matching on packet boundaries. A hardware based TCP reassembler, like the one described in Chapter 7 can also be used to reassemble TCP streams.

However a packet based pattern matching approach is sufficient for analyzing the traffic mix of a network, since most traffic is not malicious and most people do not try to evade. The goal of such an application is to determine what application layer protocols contribute to the total traffic observed on a network link. Recall the traditional approach using port number to distinguish protocols is not sufficient any more.

6.1 Environment

The network environment selected for application detection is the MWN, the *Munich Scientific Research Network* (Münchner Wissenschaftsnetz). It connects two major universities and affiliated research institutions to the Internet. The MWN heavily limits the amount of Peer-to-Peer file sharing traffic using the ipp2p [ipp] module for Linux netfilter [net]. Thus trying to detect these applications in the MWN is not much avail. Furthermore we also limit our analysis to TCP protocols. The applications our detector is looking for are:

- HTTP, the HyperText Transfer Protocol
- iTunes, http connections used for downloading multimedia files from Apple's iTunes Music Store
- RSYNC, the RSYNC protocol for synchronizing data archives, like ftp download servers
- RTSP, the RealTime Streaming Protocol
- SMTP, the Simple Mail Transfer Protocol
- NNTP, Network News Transfer Protocol
- IMAP, the Interactive Mail Access Protocol
- IMAP with starttls. Encrypted IMAP connections using the starttls command to initiate encryption.
- POP3, the Post Office Protocol, version 3
- SSH, the Secure Shell Protocol. SCP, the secure copy protocol is also detected by the patterns for ssh.
- FTP, the File Transport Protocol. Including passive mode data connections.
- HTTPS, SSL encrypted http connections. These connections are not identified via pattern matching, rather connections using well-known port 443 are accounted as HTTPS.
- abnormal termination, connections that are terminated abnormally by a TCP RST.

6.2 Design and Implementation

We need to select the patterns we want to use for protocol detection. The Application Layer Packet Classifier for Linux project [L7f] has an extensive collection of patterns for application layer protocols detection that are freely to available. The RFCs specifying protocols are another source of patterns for application detection. The patterns used for this thesis are based on the patterns from the Application Layer Packet Classifier for Linux project and patterns derived from RFCs.

In order to determine the application layer protocol spoken within a connection, the connection state must be held. The application detector uses a hash based connection table to track connections and keep their state. For every incoming packet the appropriate connection record is located and the match information of the packet is read. If a pattern matches, a counter in the connection record is incremented. When the connection is finally terminated and the connection record is evicted from the connection table, the match counters are analyzed to determine which application layer protocol the connection used. For HTTP and FTP the application detector also determines if the connection used a well known port or not. Furthermore the well known ports for these protocols are checked to find connections that use these well known ports but that do not speak the application layer protocol normally associated with that port. Eviction of connections from the connection table is solely based on an idle timeout. TCP control flags are not used.

HTTPS connections and connections that terminated abnormally are not identified by using pattern matching. Since HTTPS connections are encrypted by nature, using pattern matching is not feasible, but nevertheless HTTPS connections using the well known port of 443 account for a significant amount of total traffic. Therefore these connections are identified by their port.

FTP data connections are handled differently. If a FTP control connection is detected, the connection is parsed to determine the IP and port addresses of the oncoming data connection. This information is then stored and when a new connection arrives, it is checked against the stored information of pending FTP data connections. If the IP and ports are found, the connection is flagged as FTP data connection.

6.3 Limitations

Although the MWN is connected to the Internet using a 10 Gbps line, the available monitoring port is only 1 Gbps. At peak times, bandwidth utilization exceeds this 1 Gbps limit. Even if the bandwidth is slightly lower than 1 Gbps the switch's monitoring port might not be able to handle all packets. Therefore it is possible that some connections cannot be classified because some packets of it are missing. Indeed, we experienced HTTP connections, that were not classified as HTTP. For

Protocol	num conns	num pkts	volume [GB]	volume [%]
HTTP	65836978	6635431736	4983	54.0%
RSYNC	3353	8677947	8	0.1%
RTSP	17076	237144513	195	2.1%
HTTPS	7030145	363987663	133	1.4%
RST	16803477	1562752776	944	10.2%
NNTP	2381	19622304	13	0.1%
SMTP	7370504	230923776	88	1.0%
SSH	4090088	1494978597	1105	12.0%
POP3	169121	143408999	83	0.9%
IMAP	87449	68735463	36	0.4%
IMPS	46718	20257981	15	0.2%
FTP	411512	283577276	193	2.1%
other	80680446	2693226526	1427	15.5%

Figure 6.1: TCP Traffic mix on the MWN using pattern matching for protocol detection

example in one case, the first server reply packet was missing on the monitor port, but deducting from TCP sequence and acknowledgement numbers, the packet was seen by the HTTP client.

For our final measurement the application detector was started on a Sunday afternoon and run for two days. Since the application detector was only run for a short period the results are not necessarily representative. The time period is sufficient however to prove the feasibility of our approach. The results returned from the application detector have not been analyzed deeply. Such analysis would enable us to refine the patterns to reduce false classifications. However a full analysis of error rates and of pattern quality would require several iterations of refining patterns and measuring the new pattern for several days, which goes beyond the scope of this thesis.

6.4 Results

During first test runs of the application detector, a lot of connections could not be identified. After analyzing the connections that could not be identified, we refined the patterns used for protocol detection and we also added more protocols which lead to the protocol catalogue of Section 6.1.

Figure 6.1 shows the results of the application detector run. The largest amount of traffic is HTTP, which accounts for more than 50% of the total traffic volume. Of the 4983 GB of HTTP traffic, 42 GB were caused by downloads from Apple's iTunes Music store and 163 GB of the HTTP traffic was using a port other than the well known port 80. Furthermore the application detector also recorded 58 GB of traffic on port 80 that was not classified as HTTP.

SSH connections also amount for a significant amount of network traffic, as could be expected from a network connecting scientific institutions to the Internet. FTP is also responsible for a significant amount of traffic, namely 193 GB. 22 GB of those were FTP connections using a port other than the well known port 21. It must be noted however, that we also detected approximately 280,000 connections on port 21, that were not detected as FTP. It must be assumed that at least some of these connections are indeed valid FTP sessions, that were not detected by the patterns used. A deeper analysis of this issue should be conducted in the future. Since FTP is used for bulk data transfers, better FTP detection will also lead to a less unclassified traffic volume.

The large traffic volume caused by connections terminated with a TCP RST flag is noticeable. Please also note, that these are only connections that could not be classified before the RST was seen. If a connection can be classified it is accounted towards the appropriate application layer protocol whether it is terminated normally or by a RST. We have not analyzed the reason for this high number of connections terminated abnormally, since this would go beyond the scope of this thesis. An analysis of these connections is an interesting field for further research however.

7 Primitive TCP Stream Reassembly

In order to facilitate hardware pattern matching for security applications like NIDS a more sophisticated approach must be taken. NIDS must be able to match patterns across packet boundaries, i.e., they must be able to do pattern matching on reassembled TCP streams. Dharmapurikar and Paxson have presented a design for a robust, hardware based TCP reassembler in [DP05]. For the task at hand however, the design presented by them is not feasible. Their reassembler is designed to be able to reassemble all TCP streams with up to one gap (due to reordering or packet loss), but to do so requires extensive bookkeeping of TCP states, implementing TCP state machines and using reassemble buffers in DRAM. All together this is too costly in terms of complexity and resource requirements.

The goal for TCP reassembly with regard to hardware pattern matching is to be able to match across packet boundaries. Furthermore the hardware reassembly does not necessarily have to reassemble *all* TCP streams. It can be argued, that reassembling the bulk of TCP connections and running pattern matching on these connections is sufficient, since the software application utilizing the pattern matching can be used to reassemble and match connections that could not be handled by the hardware. If handling all non reassembled connections is too expensive, an application could randomly select some of the non reassembled connections for analysis and thus making it impossible for an attacker to predict if his connection will be matched or not. The hardware still does pattern matching on single packets for non reassembled connections, so the software application only has to search for pattern matches on *packet boundaries* of these connections.

The TCP reassembler presented here, is able to reassemble TCP connections that are received in order without any gaps and that are carried in unfragmented IP packets. All other situations have to be handled by the application software. According to Dharmapurikar and Paxson [DP05] reordering occurs only in $2 - 3\%$ of all TCP traffic, while Jaiswal et al. [JID+03] found that reordering occurs in $3 - 5\%$ of all TCP traffic. Therefore handling connections with reordering in software is feasible. IP fragmentation is not an issue, since all major TCP stacks set the *Don't Fragment* flag for IP packets, so the number of fragmented IP packets carrying TCP data is low.

Reassembling only in-order TCP connections enables a hardware reassembler design which only needs small reassembly buffers and which does not have to keep extensive state for TCP connections, like sequence numbers. Having only small reassembly buffers makes it feasible to use SRAM memory for the buffers.

The TCP reassembler requires external components. A TCAM is used to hash TCP/IP connection tuples and a SRAM is used for reassembly buffers. DAG Co-Processor cards feature an 4.5 Mbit CAM and a 18 Mbit SRAM (approximately 2 MB). When using the TCAM in a 144bit×32k entries configuration, the reassembler will be able to keep state for 32,000 parallel TCP connections. Both directions of a connection are mapped to the same CAM entry to maximize TCAM utilization. Of course, the directions have to be distinguished for the actual reassembly operation.

The design is not verified in hardware. The TCAM interface and the logic to timeout old connections is not implemented yet. The other parts of the TCP reassembler are implemented in VHDL and simulate successfully. For the simulation a software CAM is used to simulate the complete TCP reassembler. Applying the approximate pattern matcher to reassembled TCP streams is working in simulation too. The VHDL code was implemented for synthesis and should therefore be synthesizeable.

8 Conclusion

This chapter briefly summarizes this thesis and gives an idea of what might be interesting in the future in the field of hardware pattern matching for network traffic analysis.

8.1 Summary

In this thesis we have analyzed possibilities for hardware based pattern matching in FPGAs to speed up applications in the field of network traffic analysis. Various algorithms for pattern matching have been analyzed and their suitability for hardware implementation has been assessed. Two approaches were chosen for implementation — a Rabin-Karp based exact pattern matcher and an approximate pattern matcher with limited expressions. The implementation was done on Endace DAG network monitoring cards, which features FPGAs from the Xilinx VirtexII and Virtex4 families.

The implemented pattern matchers were analyzed and evaluated and we showed, that using hardware for pattern matching is feasible and can lead to significant speed improvements over pure software solution. To further demonstrate the applicability of hardware pattern matching, an example application for protocol analysis and accounting was developed and used to analyze the traffic mix on the Internet connection of the MWN, the *Munich Scientific Research Network* (Münchner Wissenschaftsnetz).

Furthermore the importance of TCP stream reassembly in conjunction with pattern matching was discussed. We analyzed the complexity of TCP reassembly in hardware and found that it is feasible to use a system, that only reassembles in-order TCP streams in hardware and handles the remaining TCP streams in software.

8.2 Outlook

An interesting project would be the integration of the hardware pattern matcher with software applications, that do pattern matching in software like the Bro NIDS. Especially the dynamic protocol detection mechanism and the IRC botnet detection mechanism [DFM+06] will benefit from hardware pattern matching. It would also be interesting to analyze what combination of Rabin-Karp and approximate pattern matching is best. Are two separate implementations good or should both matchers be combined in one system resp. one FPGA? How many FPGA resources should be allocated to the Rabin-Karp matcher and how many to the approximate pattern matcher?

The integration of hardware based TCP reassembly together with hardware pattern matching is an interesting field for further investigation too. What will be the benefits of the proposed TCP reassembler? Using a different approach for TCP reassembly might also yield interesting results.

The application detector framework sketched in Section 6 also shows paths for future research. The reason for the large amount of TCP connections that terminated abnormally can be analyzed. Furthermore the application detector itself can be enhanced to a fully fledged application, which requires reviewing and profiling the pattern sets and application layer protocol catalogue, that we want to match.

A Glossary

ASIC	Application Specific Integrated Circuit
BlockRAM	16kbit memory modules availabe in Xilinx FPGAs
CAM	Content Addressable Memory
DRAM	Dynamic Random Access Memory
EOD	End Of Data. A one bit signal send along with a datapath to indicate the last word of data of a packet.
ERF	Exentsible Record Format
FPGA	Field Programmable Gate Array
IDS	Intrusion Detection System
ISP	Internet Service Provider
LUT	Lookup Table, one of the internal building blocks of FPGAs
NIDS	Network Intrusion Detection System
RFC	Request for Comments
SDRAM	Synchronous DRAM
slice	One of the internal building blocks of blocks of FPGAs. A slice contains 2 LUTs and 2 Flip Flops
SOD	Start Of Data. A one bit signal send along with a datapath to indicate the first word of data of a packet.
SRAM	Static Random Access Memory
TCAM	Ternary Content Adressable Memory

B Short Introduction to FPGAs

This chapter gives a very short introduction to FPGAs. Another good introduction to FPGAs can be found in the Wikipedia [Wik07b].

FPGA is the abbreviation for Field Programmable Gate Array. It contains programmable logic blocks and a programmable interconnection network between the logic blocks. An FPGA is in general slower than an ASIC, but it has the advantage of being programmable in the field. I.e., a FPGA can reprogrammed after the circuit board or appliance containing the FPGA has been deployed.

The logic blocks inside a FPGA are called *slices*. These slices can be interconnected pretty much at will using the interconnection or routing network. The main components on a slice in a Xilinx FPGA are two FlipFlops and two LUTs. FlipFlops are the storage elements resp. registers required for synchronous logic design. The LUTs are used to implement combinatorial logic functions. A LUT has four inputs and one output and can be used to implement any logic function or truth table with up to four inputs. Besides LUTs and FlipFlops a slice also contains other components, like multiplexers, carry-chains, etc. The calculation of the carry bits during an arithmetic operation, like an addition, is quite time consuming. Since arithmetic operations are rather common and using LUTs to implement the carry logic results in long propagation delays, dedicated logic to calculate carry bits, the *carry chains*, are embedded on the slices. A Xilinx VirtexII-2000 FPGA for example contains about 10750 slices.

Next to slices and the routing network FPGAs also contain other logic blocks in lower numbers. For example Xlinix VirtexII FPGAs have dedicated SRAM resources, called BlockRAMs, and dedicated multiplier circuits. Some modern FPGAs also contain highly integrated logic blocks, like CPU cores, Ethernet transceivers and PCI cores.

Logic design for FPGAs is in general done using a Hardware Description Language. The most common examples of such languages are VHDL and Verilog. The tool chain to translate a VHDL or Verilog source file into an image that can be loaded into a FPGA is roughly as follows:

synthesis The source code is transformed into fundamental logic functions, like FlipFlops and combinatorial blocks.

map The output from the synthesis phase is taken and mapped onto the logic resources (LUTs, BlockRAMs, etc.) available in the FPGA.

place-and-route The logic blocks are finally placed and interconnected. The place-and-route phase must also ensure, that timing constraints are met.

Originally VHDL was developed for logic simulation and verification. Therefore VHDL allows language constructs, that work perfectly well in simulation but that cannot be translated into hardware. VHDL code that can be translated into hardware is called *synthesizeable*, while other code, is called *not synthesizeable.*

Bibliography

For an easy lookup of citations an Internet link to the location were the document or the application was found is supplied. Please be aware of the fact the the cited Internet contents may change. Therefore, the date in parentheses tells the point in time on which the URL was last accessed.

[BM77] Robert S. Boyer and J. Strother Moore. A lemma driven automatic theorem prover for recursive function theory. In *IJCAI*, pages 511–519, 1977.

[BS06] S.A. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. In *Proceedings of IEEE Infocom*, 2006.

[CS03] Christopher R. Clark and David E. Schimmel. A pattern-matching co-processor for network intrusion detection systems. In *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 68–74, Tokyo, Japan, 2003.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[DFM+06] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *Proceedings of the 15th Usenix Security Symposium*, pages 257–272, 2006. `http://www.net.informatik.tu-muenchen.de/~hdreger/papers/USENIX_SEC06-DynAppAnalysis.pdf` (on 27 Feb 2007).

[DP05] S. Dharmapurika and V. Paxon. Robust TCP stream reassembly in the presenece of adversaries. In *Proceedings of the 14th Usenix Security Symposium*, 2005. `http://www.icir.org/vern/papers/TcpReassembly/TcpReassembly.pdf` (on 27 Feb 2007).

[End04] Endace Meassurment Systems, Auckland, New Zealand. *ERF – Endace Extensible Record Format*, June 2004. `http://www.endace.com/support/EndaceRecordFormat.pdf` (on 22 Feb 2007).

[End07] Endace Ltd. Homepage, 2007. `http://www.endace.com` (on 02 May 2007).

[IANA] Internet Assigned Numbers Authority. `http://www.iana.org/` (on 25 Feb 2007).

[ipp] ipp2p project homepage. `http://www.ipp2p.org/` (on 02 May 2007).

[JID+03] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. Measurement and classification of out-of-sequence packets in a tier-1 ip backbone. In *Proceedings of the IEEE Infocom*, San Francisco, March 2003.

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. `http://www.research.ibm.com/journal/rd/312/ibmrd3102P.pdf` (on 21 Feb 2007).

[L7f] Application layer packet classifier for linux. `http://l7-filter.sourceforge.net/` (on 25 Feb 2007).

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710, 1966. Original in *Doklady Akademii Nauk SSSR* 163(4): 845–848 (1965).

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. `http://www.egeen.ee/u/vilo/edu/2002-03/Tekstialgoritmid_I/Articles/Approximate/Navarro_Review_on_Approximate_Matching_p31-navarro.pdf` (on 26 Jan 2007).

[net] Linux netfilter project homepage. `http://www.netfilter.org/` (on 02 May 2007).

[Pax99] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24), December 1999. `ftp://ftp.ee.lbl.gov/papers/bro-CN99.ps.gz`.

[PCR] PCRE library project homepage. `http://www.pcre.org/` (on 02 May 2007).

[Roe99] Martin Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proc. 13th Systems Administration Conference - LISA '99*, pages 229–238, 1999.

[Sel80] Peter H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.

[SP01] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01)*, April 2001.

[SP03] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2003.

[TRE] TRE library project homepage. `http://laurikari.net/tre/` (on 02 May 2007).

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985.

[Wik07a] Wikipedia. Boyer-Moore algorithm. `http://en.wikipedia.org/w/index.php?title=Boyer%E2%80%93Moore_string_search_algorithm&oldid=119109024` (on 19 Apr 2007), 2007.

[Wik07b] Wikipedia. Field Progammable Gate Array (FPGA). `http://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array\&oldid=109298228` (on 23 Feb 2007), 2007.

[Wik07c] Wikipedia. Knuth-Morris-Pratt algorithm. `http://en.wikipedia.org/w/index.php?title=Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm&oldid=119212834` (on 19 Apr 2007), 2007.